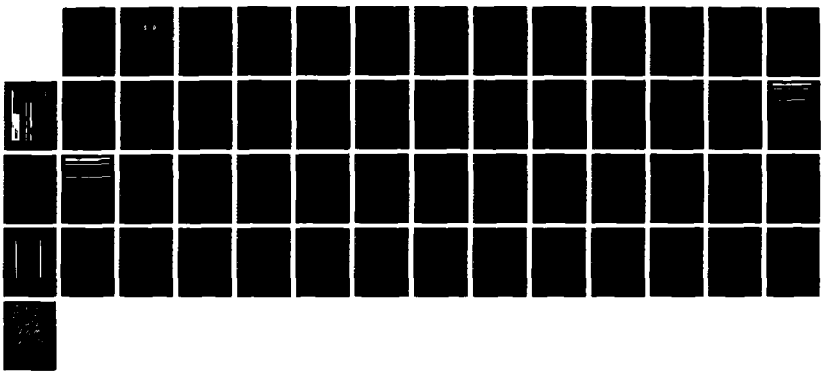
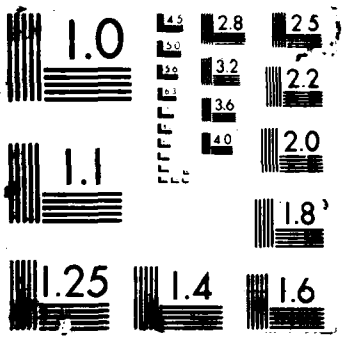
AD-A194 920 THE BBN (BOLT BERANEK AND NEWMAN) KNOWLEDGE ACQUISITION 1/1
PROJECT PHASE 1 F..(U) BBN LABS INC CAMBRIDGE MA
G ABRETT ET AL. MAY 87 BBN-6543 F30602-85-C-0085
UNCLASSIFIED F/G 12/5 NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0

BBN Laboratories Incorporated

A Subsidiary of Bolt Beranek and Newman Inc.

Report No. 6543

The BBN Knowledge Acquisition Project: Phase 1—Final Report; Functional Description; Test Plan

Glenn Abrett and Mark H. Burstein

May 1987

Prepared for:
Defense Advanced Research Projects Agency

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: BBN Report No. 6543	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): THE BBN KNOWLEDGE ACQUISITION PROJECT: PHASE I- FINAL REPORT; FUNCTIONAL DESCRIPTION; TEST PLAN		5. TYPE OF REPORT & PERIOD COVERED: Phase I - Final Report
		6. PERFORMING ORG. REPORT NUMBER: BBN Report No. 6543
7. AUTHOR(s): Glenn Abrett and Mark H. Burstein		8. CONTRACT OR GRANT NUMBER(s): F30602-85-C-0005
9. PERFORMING ORGANIZATION NAME AND ADDRESS: BBN Laboratories Inc. 10 Moulton St. Cambridge, MA 02238		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: Defense Advanced Research Projects Agency 1400 Wilson Blvd. Arlington, VA 22209		12. REPORT DATE: May 1987
		13. NUMBER OF PAGES: 48
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report): Unclassified
		15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution of this document is unlimited. It may be released to the Clearinghouse, Department of Commerce, for sale to the general public.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Knowledge acquisition, knowledge editing, knowledge representation, expert systems, strategic computing.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This document presents the final report on Phase I of KREME, BBN's Knowledge Representation, Editing and Modeling Environment. It includes a brief functional description of KREME, a test plan that describes how to load and boot KREME, and two sample KREME networks. KREME was designed to handle large knowledge bases, support experiments with knowledge engineering techniques, and implement a usable system for knowledge acquisition and maintenance. This document describes KREME's frame editor, macro editor, classifier, knowledge integrator, and rule editor.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Report No. 6543

THE BBN KNOWLEDGE ACQUISITION PROJECT: PHASE 1 - FINAL REPORT; FUNCTIONAL DESCRIPTION; TEST PLAN

Glenn Abrett and Mark H. Burstein

May 1987

Prepared by:

BBN Laboratories Incorporated
10 Moulton Street
Cambridge, Massachusetts 02238

Prepared for:

Defense Advanced Research Projects Agency
1400 Wilson Boulevard
Arlington, Virginia 22209

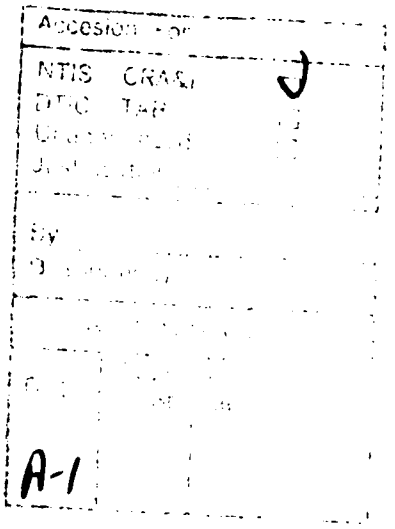


Table of Contents

List of Figures

1. Introduction

This is the Final Report for Phase One of the BBN Laboratories Knowledge Acquisition Project. This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and was monitored by the Rome Air Development Center (RADC) under contract number F30602-85-C-0005.

The goal of this project was to create a useable and extensible knowledge engineering environment that will be capable of handling very large knowledge bases, support experiments with knowledge engineering techniques and implement a useable system for knowledge acquisition and maintenance. During Phase One of this project we have created the KREME Knowledge Representation Editing and Modeling Environment. KREME is an extensible experimental environment for developing and editing large knowledge bases in a variety of representation styles. It provides tools for effective viewing and browsing in each kind of representation, automatic consistency checking, macro-editing facilities to reduce the burdens of large scale knowledge base revision and some experimental automatic generalization and acquisition facilities.

Among the planned extensions to KREME are:

- The Procedure Editor
- A KEE Interface
- The Addition of Boolean Connectives to Slot Restrictions
- Extension of the Macro Editor

We are currently in the process of extending the value restriction language to permit more complex forms containing conjunctions, disjunctions and negations, based on the restriction language for KEE[tm][1] frames [6]. This effort should result in an extended classifier, as well, capable of maintaining consistency among frames in the KEE class of frame languages.

During Phase Two we will also be developing experimental kinds of automatic knowledge acquisition: techniques for generating controlled acquisition dialogues, procedures to automatically transform previously acquired knowledge for use in new tasks, and techniques for learning by analogy and from examples.

[1] KEE is a trademark of IntelliCorp

2. Overview of the BBN Knowledge Acquisition Project

Our goal has been to develop an environment in which the problems of knowledge acquisition faced by every knowledge engineer attempting to build a large expert system are minimized. We believe both knowledge engineers and subject matter experts with some knowledge of basic knowledge representation techniques will find it easy to use KREME to acquire, edit, and view from multiple perspectives knowledge bases that are several times larger (i.e., 5-10,000 concepts) than those found in most current systems.

KREME attempts to deal with the inextricably related problems of knowledge representation and knowledge acquisition in a unified manner by organizing multiple representation languages and multiple knowledge editors inside of a coherent global environment. A key design goal for KREME was to build an environment in which existing knowledge representation languages, appropriate to diverse types of knowledge, could be integrated and organized as components of a coherent global representation system. The current KREME Knowledge Editor can be thought of as an extensible set of globally coherent operations that apply across a number of related knowledge representation editors, each tailored to a specific type of knowledge. Our approach has been to integrate existing frame and rule representation languages in an open ended architecture that allows the extension of each of these languages. In addition, we have provided for the incorporation of additional representation languages to handle additional types of knowledge.

Our approach to consistency maintenance has been to develop a *knowledge integration* subsystem that includes an *automatic frame classifier* and facilities for inter-language consistency maintenance. The frame classifier automatically maintains logical consistency among all of the frames or conceptual class definitions in a KREME frame base. In addition, it can discover implicit class relationships, since it will determine when one definition is logically subsumed by another, even when the knowledge engineer has not explicitly stated that relationship. The inter-language consistency maintenance facility checks for inconsistencies in references to frames in knowledge bases specified using other representation languages (e.g., rules, procedures).

A second important area of investigation in developing the KREME editing environment has been the attempt to provide facilities for large-scale revisions of a knowledge base. Our experience indicates that the development of an expert system inevitably requires such systematic revisions of the developed representation. This is often caused by the addition or redefinition of a task the system is to perform. These kinds of systematic changes to a knowledge base generally require painstaking piecemeal revision of each affected element, one at a time. Our initial approach has been to provide a *macro-editing* facility, in which the required editing operations can be demonstrated by

example and applied to specified sets of knowledge structures automatically. A library of generic macro-editing operations for the most common and conceptually simple (though potentially difficult to describe) operations will be developed during Phase Two of the project.

Finally, we have begun to investigate techniques for *automatic generalization* of concepts defined in a knowledge base. We will briefly describe these experiments as well, in Section 8.

Underlying the entire KREME system is a strong notion of meta-level knowledge about knowledge representation and knowledge acquisition. The representation languages were implemented based on a careful decomposition of existing knowledge representation techniques and implemented as combinable objects using FLAVORS [7]. By organizing this meta-level knowledge base modularly, behavioral objects implementing such notions as inheritance and subsumption could be "mixed in" to a variety of representational subsystems making the incorporation of new representations and their editors reasonably straightforward. That is, each object in the meta-knowledge base encodes some aspect of a traditional representational technique, and is responsible for its own display, editing and internal forms.

3. The KREME Knowledge Representation Editing and Modeling Environment

3.1 Functional Description

The KREME family of knowledge editors currently consists of three major editor modules: a frame editor, a rule editor, and a procedure editor.[2] (See figure 3-1.) KREME also includes a large toolbox of editing techniques that are shared among the editor modules. This section will describe the global environment and toolbox, later sections will describe the individual editors. Sections 3.3 through 3.5 provide a discussion of the user interface. Readers who require more detail should consult *KREME: A User's Introduction*, BBN Report No. 6508.

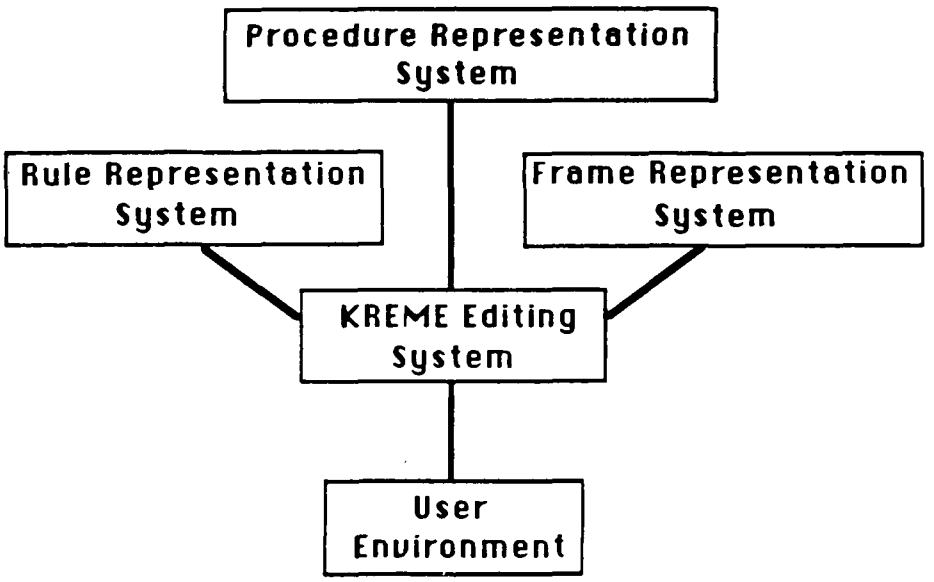


Figure 3-1: KREME: Functional Description

[2]The Procedure Editor is introduced briefly in this report. Full development of the Procedure Editor will occur in Phase Two.

3.2 Basic Editing Environment

Each type of representation included in the system has defined for it one or more editor *views*. A view is a collection of windows appearing together on the screen. Each window displays some aspect of the particular piece of knowledge being edited and/or a set of editing operations on it. When the user desires to enter or edit a specific piece of knowledge, the system opens the most appropriate view for the type of knowledge and the editing operation requested. Typically, any aspect of the knowledge being edited can be changed or viewed in more detail simply by pointing at it. This organization allows knowledge to be viewed by the user from multiple perspectives and at more than one level of detail.

The editor maintains a level of indirection between the knowledge being edited and the representation of that piece of knowledge in the knowledge base. This is accomplished by a mechanism like that of text editor buffers. Changes are always made to *editor definition objects* which are distinct from the corresponding objects in the actual knowledge base. The stack or list of the active definition objects is always visible to the user. The top item in this list is the definition currently being viewed and edited. The user is free to modify the current definition in any way without directly affecting the knowledge base. Only when the modified definition is to be placed into the knowledge base is a defining function appropriate to the type of knowledge (e.g. classification for concepts and roles), executed and the knowledge base modified.

Since the editor stack is always visible, it provides one convenient method for browsing. The user may point at any definition item currently in the stack. The object will then be displayed in the same editor view as when it was last edited.

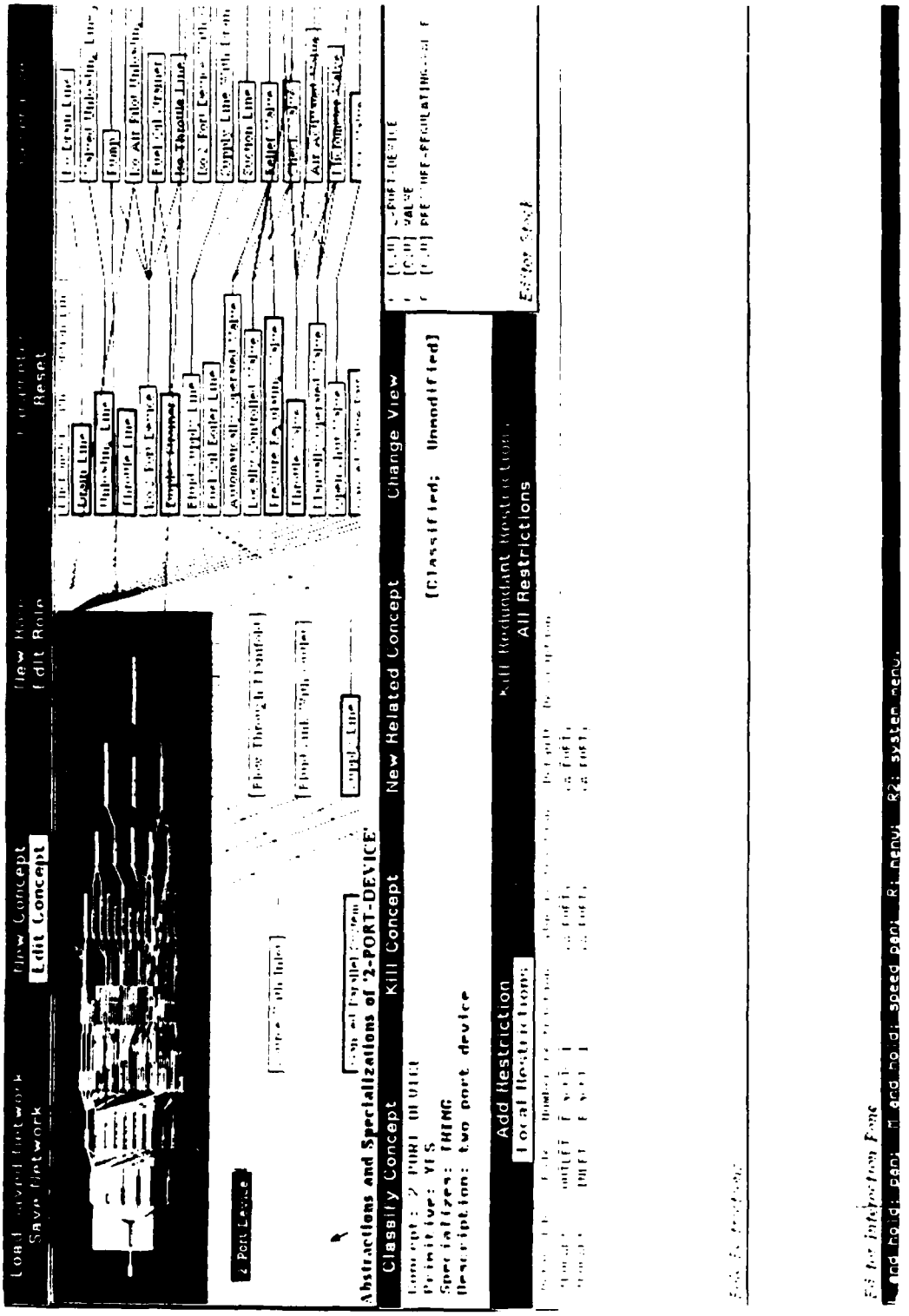
A number of window subsystems or tools have been developed and incorporated into the KREME editor to make editing, viewing and browsing in knowledge bases easier and faster. They are described below.

3.3 The Grapher

KREME is equipped with a general graphing facility that rapidly draws lattices of nodes and links. Its main use is to provide a dynamically updated display of a concept or role and its place in the specialization or inheritance hierarchy. When editing a concept in the **Main Concept Editing View** or the **Big Concept Graph View**, or when editing a role, KREME automatically displays all of that object's abstractions and specializations. More abstract objects are displayed to the left of the current editor object, and more specialized objects to the right.

As shown in figure 3-2, the current editor object appears as a black node with white letters. All other objects appear as nodes with a white background. Objects that are defined as *primitive* are indicated by bold-edged boxes. Nodes that have been ***modified*** (edited but not reclassified) have a grey background.

Figure 3-2: The Main Concept Editing View



3.3.1 Panning the Graph

The grapher can display a graph much larger than the window through which it is viewed. To see a part of the network that is off the screen, the user points with the mouse at some point on the graph window not containing a node, holds the left button down and drags the mouse. To *speed pan*, the user holds down the middle mouse button.

3.3.2 The Overview Graph

Clicking the right button once over an empty part of the graph window will make the **Graph Operations Menu** appear. If the user clicks **overview**, a miniature version of the full lattice will appear in a black region in the upper left corner of the graph window (as in figure 3-2). This overview shows a miniature version of the full network. The visible region of the graph is indicated by a white rectangle. If the user pans with the mouse over the main graph window, this white rectangle will follow the mouse movements. All of the mouse operations available on nodes in the main window will also work on nodes in this window. The name of the node being pointed at is indicated in the **mouse documentation window**. The overview window also can be used to pan the main graph window. The overview is turned off by bringing up the **Graph Operations Menu** and clicking the command **overview**.

3.3.3 The Graph Operations Menu

The other options in the **Graph Operations menu** shown in figure 3-3 are:

- **hardcopy** - Sends a copy of the full graph of the lattice to the printer.
- **style menu** - Allows the user to choose the font style and size of characters used for nodes on the graph. Smaller fonts are useful to see more of large networks at once.
- **find node** - Prompts for the name of an object on the graph, and centers that node on the graph window. It also draws a circle around the node so that the user can find it more easily. The circle disappears as soon as the graph is panned.
- **overview** - Switches the overview graph between visible and invisible.
- **orientation** - Switches the orientation of the graph. Normally, the lattice is drawn from left to right. This command will cause the graph to be redrawn from the top of the screen down, and vice versa.
- **speed pan** - This command pops up the speed panning box without having to hold down the mouse button. In this mode, clicking any mouse button will make it go away.
- **redraw graph** - Redraws the current graph.

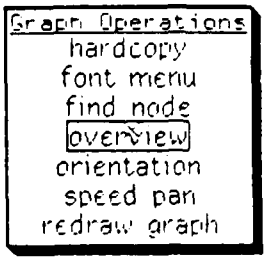


Figure 3-3: The Graph Operations Menu

3.3.4 The Graph Node Command Menu

Normally, the KREME Grapher displays only the abstractions and specializations of the current editor object, because KREME was designed to work with the very large lattices characteristic of *very large knowledge bases*. The Grapher provides a number of options to enable users to tailor the display to see more (or less) than KREME normally displays on such graphs.

Whenever the mouse is over a node on a graph, the **mouse documentation window** shows the name of the node, followed by:

```
L:Edit this node.  M:Graph Relatives  R:Menu of Editing Options
```

Clicking the left mouse button causes KREME to make the object pointed to the top editor stack item. This is an extremely convenient way of browsing through large concept networks quickly, and focusing on different portions of such a network. If, however, the user wishes to continue editing the concept that he is currently viewing, but see more (or less) of the network around that concept or some other concept *on the same graph*, he can use the **Graph Relatives Menu** found by clicking the middle mouse button over any graph node.

The **Graph Relatives Menu**, exposed by clicking the middle button over a node, contains the following commands:

- **Graph Parents** - causes all abstractions of the node clicked on to be added to the displayed graph.
- **Graph Children** - causes all specializations of the node clicked on to be added to the displayed graph.
- **Hide Children** - causes all specializations of the node clicked on to be removed from the graph, unless they are also children of some other node.

- **Hide Node and Children** - causes the node clicked on and its children to be removed from the graph.

3.3.5 Editing a Network from a Graph

Clicking the right button over a graph node causes yet another menu of options to be exposed, the **Concept Graph Edit Options Menu**.[3]

This menu contains the following options for concepts:

- **Show Definition** - This option causes the textual (LISP) form of the concept's definition to be displayed over the Graph Window.
- **Kill Concept** - This causes the concept pointed to to be removed from the knowledge base. It has the same effect as the **Kill Concept** command in the **local command menu window**, except that it works when the user is not currently editing the concept he wishes to kill.
- **Rename Concept** - This command prompts for a new name for the concept pointed to, and immediately replaces all references to that name with the new name throughout the knowledge base.
- **Delete Parent** - This command prompts for the name of a parent and then deletes that parent from the list of defined parents of the concept initially pointed to. It also switches KREME to editing the concept modified, so that it can then be reclassified.
- **Add Parent** - This command also prompts for a parent, adds the concept named to the list of defined parents of the concept, and switches to editing the modified concept.
- **Splice Out Parent** - This command prompts for a parent, and removes that parent from the list of defined parents of the concept, replacing it with *that* concept's parents. Again, the editor is switched to a view of the modified concept.

3.4 Editing in the State Window

The **state window** of the **Main Concept Editing View** displays basic information about the concept currently being edited. The top line displays the name of the concept, and any *synonyms* or alternate names for that concept. The name of the concept can be changed by clicking on the word **Concept:** and entering a new name.

The second line of the display shows whether the concept is defined as *primitive* or not, and whether the concept has been classified or modified since classification. Clicking on the word **Primitive:** causes the concept to be marked primitive if it was not, and vice versa.

The third line displays the both the *direct and defined parents* of the concept, after the word **Specializes:**. *Defined parents* are concepts that the user specifies as abstractions of the concept. *Direct parents* are concepts that may or may not have been *defined* as parents of the current one, but have been determined by the classifier to subsume the class denoted by this concept *and not have any specializations that also subsume this concept*. On the Concept Graph, the direct parents of a concept are the ones with direct links to it.

[3]On graphs of roles, the **Role Graph Edit Options Menu** appears, with essentially the same commands for roles, except as noted.

This **Specializes:** list should be read as follows: Concepts that are unmarked are both **defined parents** and **direct parents**. Concepts that are **defined parents** but not **direct parents** are prefixed by a "-". Concepts that are **direct parents** but not **defined parents** are prefixed by a "+". The user can easily add a parent to the set of defined parents of the concept.

3.5 Editing in the Table Edit Window

Normally, the **table edit window** in **Main Concept View** displays the set of **Local Slots** of the concept, that is, those slots which are defined locally by this concept and not inherited from above. The columns in the table are labeled **"Defined by", "Role", "Number Restriction", "Value Restriction", "Default", and "Description"**.

Clicking (with the left mouse button) on the command **All Slots** in the **table edit command window** causes KREME to display both local and inherited slots. In this display, local slots are indicated by the word ***LOCAL*** in the "**Defined by**" column of the table. Slots inherited from a parent show the name of that parent. Slots formed by combining the value restrictions and/or number restrictions of several parents are indicated by the word ***CLASSIFIER***. When the table window is displaying all of the concept's slots, the user can return to viewing just the local ones by clicking the command **Local Slots**.

Whenever the Table Edit Window shows slots of the current concept, the user can edit those slots or add new ones. To change the slot name, value restriction, number restriction, default, or description of a slot, the user simply clicks the left mouse button over the thing to be changed, and will be prompted for a replacement. For all but number restrictions, the right button will pop up a menu that includes the commands: **Change** the part of the slot pointed to, **Show Definition** of the concept or role pointed to, **Edit Definition** of that concept or role, or pop up a **Graph** of its abstractions and specializations. When pointing to the slot name, in the column labeled "**Role**", the user can also **Rename Role**, that is, change the name of the role, and all references to it in the knowledge base.

When the mouse is over a line in the slot table, and the entire line is encircled by a box, the right mouse button can be used to get a menu of **Delete Slot**, **Copy Slot** to another concept, and **Move Slot** to another concept. For the last two, KREME prompts for the name for the concept to move or copy the slot to.

3.5.1 Adding New Slots

Whenever the slots table window is visible, as in the **Main Concept Editing View**, the user can add new local slot definitions. A new slot is added to the defined slots of the concept with the **Add Slot** command. When this command is issued, the system prompts for a role name, a value restriction, a number restriction and a default form. Any of these items can be entered by typing or by pointing to the desired name or form if it is visible.

If a role or concept named in a role restriction or default does not exist, the system will offer to make one with the name given, and proceed to pop up the defining form for that object. When the user is finished filling out the form, he clicks **Define**, and KREME will continue to ask for the rest of the new slot's features.

When the user has finished adding and modifying the slots of a concept, he should always make the changes permanent with the **Classify Concept** command.

3.5.2 Modifying the Table Edit Window

The appearance of Table Edit Windows can be modified in several ways. The tables are scrollable in both the up-down and left-right directions.If the user does not wish to see some columns of the table, they can be selectively removed.

3.5.3 Changing the Contents of the Table Window

Since there is not enough room in the Main Concept Editing View to display all of a concepts defining features at one time, the contents of the Table Edit Window can be changed to display those other features. To do this, the user must use the mouse to find the **table window contents menu**. This menu is available wherever there is nothing else under the mouse while still inside the table window. the user will know he has found it because the **mouse documentation window** will show the words:

```
R: Change the contents of this table.
```

When the user clicks the right button, he will see the following menu options:

- **Slots** - Displays the table of this concept's slots, as described above.
- **Inverse Restrictions** - Displays a table, essentially like the slots table, but of all of the slots displayed are slots of other concepts that use the current concept as their *value restriction*. This table is useful when tracing references to a concept in other concepts. When this table is displayed, the **table edit command window** will be empty. Some of the editing options described for the slots table will not work here.
- **Slot Equivalences** - This table displays the **slot equivalences** of the current editor concept. This table has only three columns, **"Defined by"**, **"Path 1"** and **"Path 2"**. The two paths are designated as denoting the same object. Since slot equivalences can be inherited, their source is also indicated in the table, in the column **"Defined by"**. When this table is visible, the **table edit command window** will show the commands **Local Equivalences**, **All Equivalences**, and **Add Equivalence**. The first two just change which equivalences are displayed. The last prompts for two slot paths that should be made equivalent.

- **Disjoint Concepts** - This table is just a one column list of all of the concepts that are defined to be disjoint from the one currently being edited. When this table is visible, the **Table Edit Command Window** will display the commands **Add Disjoint Class**, **Local Disjoint Classes**, and **All Disjoint Classes**.

3.6 Files and Multiple Language Support

All definitions manipulated by the editor are read and stored in lisp-readable text files of defining forms. Since these files contain formatted lisp forms, they are user-readable, and can be edited offline using an ordinary text editor. In fact, KREME can as easily read files that were developed independently using a text editor or some other frame editor.

Files are read in using the LOAD command. A file can be loaded into a blank KREME knowledge base or can be loaded on top of an already existing knowledge base. This mechanism, which relies heavily on the the frame classifier to maintain consistency, enables KREME to organize information from multiple knowledge bases to create a single unified whole.

KREME currently reads and writes definitions in either its own frame language syntax or NIKL syntax. This flexibility has made it possible for KREME to be used regularly to examine and update a knowledge base of approximately 1000 roles and concepts for the IRUS/JANUS natural language interface that was built using NIKL. KREME can also read files of MSG (the frame language of the STEAMER [21] system) defining forms, providing access to the extensive STEAMER knowledge base of concepts and procedures. We are currently building an interface to files of KEE frame definitions.

This multiple language handling facility is a crucial feature of KREME. A library of input translation programs will enable a knowledge base builder using KREME to draw upon previously existing knowledge bases to create new knowledge bases.

4. The KREME Frame Editor

This section will describe the KREME knowledge editor for a frame representation language.

4.1 The KREME Frame Language

A number of frame languages have been developed in recent years to support AI systems [11, 2, 17, 9, 3, 6, 8]. These languages have been well researched and extensively tested, and our most important criteria for a suitable frame representation language were that it:

1. Allowed multiple inheritance
2. Was a logically worked out mature language.
3. Had some mechanism for internal consistency checking.
4. Was built on a modular object oriented base so that the language could be decomposed in such a way as to make it easily extensible.

NIKL (the definitional or frame language component of KL-TWO) [9, 14, 20] seemed an ideal candidate. It is a fully worked out frame representation language that allows multiple inheritance, is reasonably expressive and, perhaps most importantly, was designed to work effectively with an automatic classification algorithm that could be easily adapted to provide a powerful mechanism for consistency checking and enforcement during knowledge base development. However, no object-oriented implementation of NIKL existed, and the NIKL classifier was not designed to allow *modification* and *reclassification* of previously defined concepts. A second frame language, known as MSG, had been built as part of BBN's STEAMER project and is object oriented in both of the above senses.

To develop KREME, we elected to reimplement NIKL as an object oriented language using MSG as a guide. The NIKL data structures were decomposed into a modular hierarchy of flavor definitions, and the KREME frame language was then built out of these flavors. This enabled us to incorporate the sophisticated instantiation mechanism of MSG with minimal effort. In the process, we were also able to implement a modular version of the NIKL classifier algorithm. This provided the kind of reclassification capability required for a knowledge editing environment and anticipated the extension of the classifier to deal with the richer semantics of languages like Intellicorp's KEE [6].

4.1.1 Frame Language Syntax

The remainder of this section will briefly describe the basic definitional syntax of the KREME Frame language. As this syntax closely resembles the formal syntax of NIKL interested readers are referred to [9] for more detail.

Following NIKL, a KREME frame is called a *concept*. Collections of concepts are organized into a rooted *inheritance* or *subsumption lattice* sometimes referred to as a *taxonomy* of concepts. A single distinguished concept, usually called THING, serves as the root or *most general concept* of the lattice. A concept has a *name*, a textual *description*, a *primitiveness* flag, a list of concepts that it *specializes* or is *subsumed by*, a list of *slots*, a list of *slot equivalences*, and a list of concepts that it is *disjoint from*

The lists of slots, slot equivalences and disjoint concepts are collectively referred to as the *features* of a concept. If each concept can be thought of as defining a unique category, then features of the concept define the necessary conditions for inclusion in that category. If a concept is not marked as *primitive*, the features also constitute the complete set of sufficient conditions for inclusion in that category.[4] A concept inherits all features from those concepts above it in the lattice (those concepts that subsume it, and, thus, are more general) and may define additional features that serve to distinguish it from its parent or parents.

Slots (sometimes called role restrictions) consist of a role or slot name, a value restriction, a number restriction and an (optional) default form. The *value restriction* specifies the class of concepts allowed as values for that slot. As in NIKL, value restrictions usually specify a particular concept.

Slot Equivalences describe slots (and slots of slots) that *by definition* must always refer to the same entities

The role name specified for each KREME slot refers to an object called a *role*. Roles in KREME, as in NIKL and several other frame languages like KRYPTON [3], and KnowledgeCraft [8], are actually distinct, first class objects that form their own distinct taxonomy, rooted at the most general possible role, usually called RELATION. Roles describe two place relations between concepts. A *role restriction* at a concept is thus a specification of the ways a given role can be used to relate that concept to other concepts.

[4]Concepts marked as *primitive* (sometimes referred to as *Natural Kinds*) have no complete set of sufficient conditions. For example an ELEPHANT must, by necessity, be a MAMMAL, but without an exhaustive list of the attributes that distinguish it from other mammals, it must be represented as a primitive concept. The class of WHITE ELEPHANTs, on the other hand, might be completely described as a ELEPHANT with slot COLOR restricted to WHITE.

4.2 Using the Frame Editor

The KREME frame editor has five views, the **Main Concept Editing View**, the **Alternate Concept Editing View**, the **Big Graph View**, and the **Macro Structure Editor View**. Roles, which are also part of the KREME Frame language, are edited with the **Role Editing View**. In this section, we will cover the details of the editing operations available in the first three of these views.

4.2.1 Editing in the Main Concept Editing View

Normally, when one creates a new concept or edits a concept for the first time, KREME makes that concept the top concept on the Editor Stack, and switches to display the Main Concept Editing View. There, KREME displays the concept as it exists at that time.

Figure 3-2 shows how the **graph window** immediately displays all of the abstractions and specializations of the concept being edited, the **state window** shows its name, whether it is primitive or not, its edit state (classified or not, modified or not), its parents, and a textual description. The **table window** simultaneously displays all of the concept's locally defined slots.

4.2.2 Frame Editing Operations

Space does not permit a full description of the functionality of the KREME frame editor so we will very briefly touch upon a few of its more important operations.

Making new concepts. The *New Concept* command in the global command menu initiates the definition of a new concept that is (1) fully specified by the user, (2) similar to some already defined concept, or (3) a specialization of one or several other defined concepts. When the initial form for the new concept has been specified the system creates a new concept definition for it and shows this new definition in the main concept view. The user is then free to add details (slots, equivalences, additional parents, etc.) to the new concept definition, classify it, or edit other concepts.

Adding and modifying slots. Whenever the window displaying slots is visible, slots can be added or modified. A new slot is added to the defined slots of the concept with the *Add Slot* command. Any portion of a slot's definition can be entered by typing or by pointing to a visible reference to the desired item. When a role or concept name that is not defined is specified, the system offers to make one with the name given.

Users may modify any locally defined slot or inherited slot. Slots shown in table windows are modified by pointing at the appropriate subform and then either typing in or pointing to a replacement form. Modifying an inherited slot causes the new definition to be locally defined.

Adding and Deleting parents. The system displays the classifier determined parents of a concept in two places. The concept graph displays them as part of the abstraction hierarchy of the concept, and the state pane indicates both the defined and direct or computed parents of the concept after the word "**Specializes:**". Since the classifier may have found that the concept being edited specializes some concepts more specific than those given as its defined parents, defined parents that are not direct parents are preceded by a "-", while classifier determined parents that were not defined parents are preceded by a "+".

Adding new defined parents to a concept's definition is done by clicking on the word "**Specializes:**" in the state window and typing a concept name or pointing to any visible concept. Parents can be deleted by clicking on their names in the list of parents displayed in the state window.

Changing names and killing concepts and roles. KREME allows the user to change the names of concepts and roles or to delete them completely. Name changing is accomplished simply by pointing at the concept or role's name in the state window and entering a new name. The *Kill* command splices a concept out of the taxonomy by connecting all of its children to all of its parents.

5. Large-Scale Revisions of Knowledge Bases

As knowledge bases grow larger, and the sets of tasks that intelligent systems are called upon to perform expands, system developers will need automatic methods for revising and reformulating accumulated knowledge bases. Toward this end, we feel that it is important to find ways of expressing *reformulations* of sets of frames and other representations and to begin developing facilities supporting the generation of new representations from old ones.

We are taking two different approaches to these problems. First, we have developed a macro facility for reformulations that can be expressed as sequences of standard, low-level editing operations. This facility allows users to use an example to define editing macros that can be applied to sets of frame definitions. Second, we are building a library of functions providing standard editing operations that cannot be defined simply as sequences of low level editing operations. Our main purpose in this project is to collect and categorize a number of different kinds of knowledge base reformulations. Our hope is that a large fraction of these operations can be conveniently described using the macro facility, as it is more accessible to an experimental user community than any set of "prepackaged" utilities, and can be more responsive to the, as yet, largely unknown special needs of that community.

5.1 The Macro and Structure Editor

One of the views available when editing concepts in KREME is the *macro and structure editor*. This view (See figure 5-1.) provides display and editing facilities for concept definitions, based loosely on the kind of structure editor provided in many LISP environments. The view provides two windows for the display of stylized defining forms for concepts. The *current edit window* displays the definition of the currently edited concept (the top item on the editor stack). The *display window* is available for the display of any number of other concepts. Any concept which is visible in either window can be edited, and features can be copied from one concept to another by pointing. Both windows are scrollable to view additional definitions as required.

There is a menu of commands for displaying and editing definitions that includes the commands **Add Structure**, **Change Structure**, **Delete Structure**, **Display Concept** and **Clear Display**. Arguments (if any) to these commands may be described by pointing or typing. Thus, to delete a slot, one simply clicks on **Delete Structure** and the display of the slot to be deleted. Adding a structure is done by clicking on **Add Structure**, the keyword of the feature class of the concept one wishes to add to (e.g., **Slot:**). The new slot itself may be copied from a displayed concept by pointing, or a new one may be entered from the keyboard. Changing (that is, replacing) a structure can be done by pointing in succession at the **Change Structure** command, the item to be replaced, and the thing to

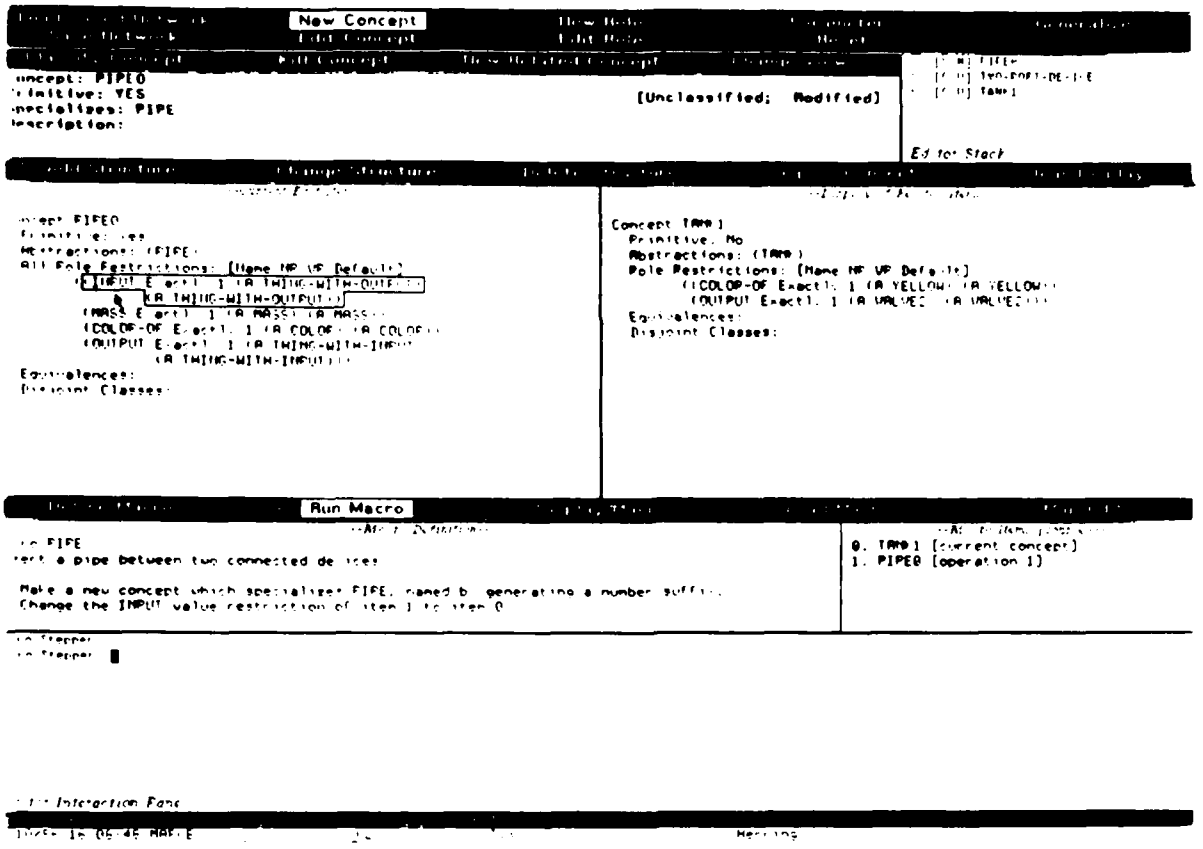


Figure 5-1: The Macro Structure Editor View

replace it with. In most cases, **Change Structure** can also be invoked simply by pointing at the structure to be replaced, without the menu command.

The last two commands in the structure view's main menu provide the means to change what is displayed in the display window. Pointing at **Display Structure** and then at any visible concept name places the definition of that concept in the display window. **Clear Display** removes all items from the display window. Individual concepts can be deleted from the display window by pointing at them and clicking. The **Edit Concept** command is used to change what is displayed in the current edit window. Editing a new concept moves the old edit concept to the bottom of the display window.

5.2 Developing Macro Editing Procedures

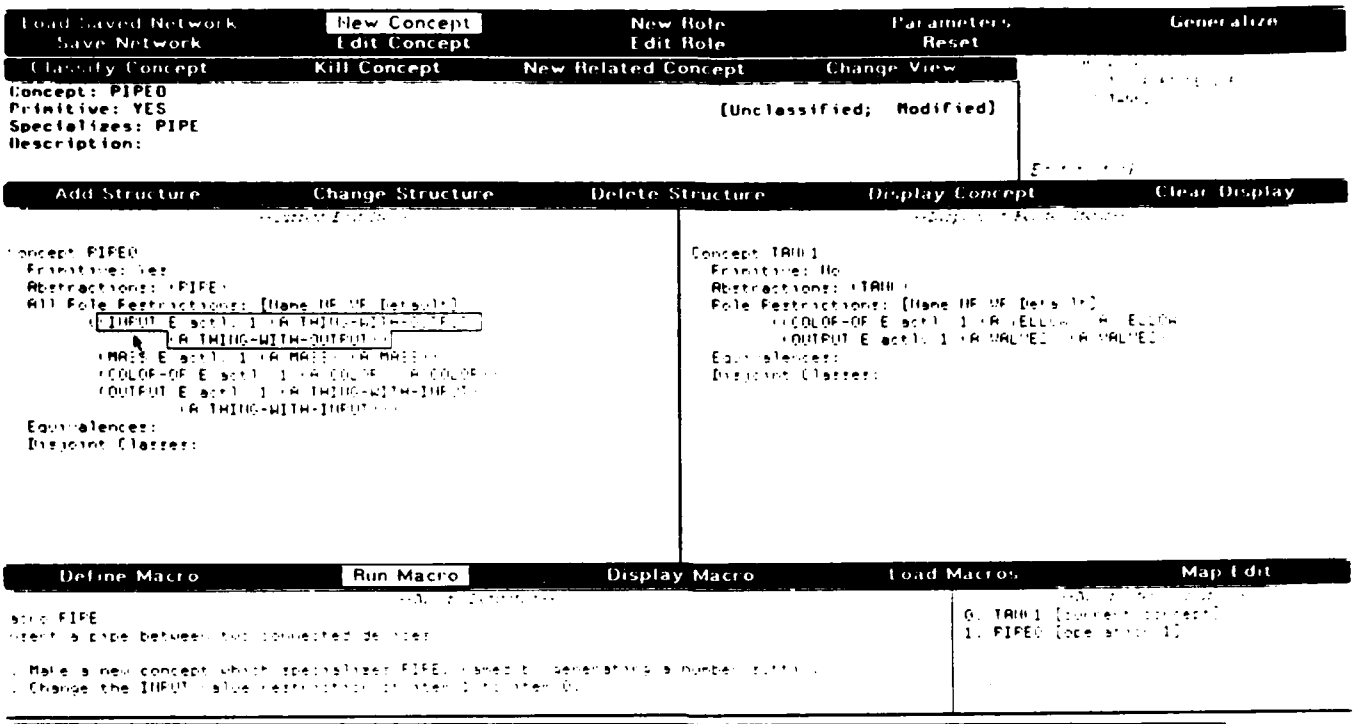
These operations, together with the globally available commands for defining new concepts and making specializations of old concepts essentially by copying their definitions, provide an extremely flexible environment in which to define and specify modifications of concepts with respect to other defined concepts. Virtually all knowledge editing operations can be done by a sequence of pointing steps using the current edit window and the display window. This style of editing is also used in the rule editor. The combination of editing features and mouse-based editor interaction style provides an extremely versatile environment for the description, by example, of a large class of editing macros.

In order to have macros, defined essentially by example, work on concepts other than those for which they were defined, the operations recorded cannot refer directly to the concepts or objects which were being edited when the macro was defined. This is handled by a kind of implicit variablization, where the objects named or pointed to are replaced by references to their relationship to the initially edited object. In most cases, these indirect references can be thought of as references to the *location* of the object in the structure editor's display windows. In fact, each new object that is displayed or edited in the course of defining a macro is placed on a stack called the *macro items list*, together with a pointer to the command that caused the item to be displayed. The utility of this form of reference will become clearer with an example.

5.2.1 Macro Example: Adding Pipes Between Components

When the STEAMER [21] system was developed, a structural model of a steam plant was created to represent each component in the steam plant as a frame, with links to all functionally related components (e.g., inputs and outputs) represented as slots pointing at those other objects. So, for example, a tank holding water to be fed into a boiler tank through some pipe that was gated by a valve was represented as a frame with an OUTPUT slot whose value was a VALVE. The OUTPUT of that VALVE was a BOILER-TANK. The pipes through which the water was conveyed *were not represented* since they had no functional value in the simulation model. If it had become important to model the pipes, e.g because they introduced friction or were susceptible to leaks or explosions, then the representational model that STEAMER relied on would have required *massive* revision. Each component object in the system would have needed editing to replace the objects in its INPUT and OUTPUT slots with new frames representing pipes that were in turn connected by their OUTPUT slots to the next component in the system.

One of our goals in developing the KREME macro editor was to be able to make such changes easily. While they are simple to describe, they normally require many tedious editing operations to a large number of concepts. Figure 5-2 shows a macro that can be applied to all objects in a system with INPUT and OUTPUT slots, in order to



While Editing TANK1:
Click on Define Macro. (*Makes Macro Item 0 = TANK1*).

1. **Make a new concept which specializes PIPE.** (*Creates PIPE0 as item 1*).
2. **Change the INPUT value restriction of item 1 (*PIPE0*) to item 0 (*TANK1*).**
3. **Change the OUTPUT value restriction of item 1 (*PIPE0*) to the OUTPUT value restriction of item 0 (*OUTPUT of TANK1 = VALVE1*).**
4. **Classify the current edit concept (*Defines PIPE0*).**
5. **Change the OUTPUT value restriction of item 0 (= *VALVE1*) to item 1 (*PIPE0*).**
6. **Classify item 0 (*TANK1*).**
7. **Edit the OUTPUT value restriction of item 1 (*Creates item 2 = VALVE1*).**
8. **Change the INPUT value restriction of item 2 (*INPUT of VALVE1 = TANK1*) to item 1 (*PIPE0*).**
9. **Classify all items.**

Figure 5-2: Steps in PIPE Macro

generate and insert PIPEs into those slots. The macro also sets the OUTPUTs of those PIPEs to be the concept that was the old value of the OUTPUT slot in the concept edited, and similarly redoes all INPUTs.

Figure 5-2 shows how the macro is defined, by editing a representation of a tank (TANK1) connected (by role OUTPUT) to a valve (VALVE2). The sequence of steps required, defined only using the mouse, is shown in figure 5-2, as they would appear in the *Macro Definition* window of the editor.

In Phase One, work on macro editing was only just begun. However, this technique already shows promise as a method for accomplishing restructurings of knowledge. We see our investigation of macro editing as only the first step in developing a knowledge reformulation facility that will make use of the higher level structure of the represented knowledge

6. Knowledge Integration and Consistency Maintenance

One of the most time consuming tasks in building large knowledge bases is maintaining internal consistency. Modification, addition or deletion of knowledge in one part of a knowledge base can have wide ranging consequences to both the meaning and structure of the knowledge stored in other parts of the knowledge base. A central component of the KREME system design was that it incorporate tools for consistency maintenance both within and across representation languages. These tools are collectively referred to as the *knowledge integrator*. When new knowledge is entered or existing knowledge modified it is the task of the knowledge integrator to propagate, throughout the knowledge base, the changes that this new or modified knowledge entails, and to report any inconsistencies that have been caused by the change.

In essence, the knowledge integrator takes each new or changed chunk of knowledge (e.g., a frame, role, rule or procedure) and determines, first, how the new definition fits into the knowledge base and, second, which other definitions depend on the current one for their meaning within the knowledge base. These dependencies are placed on an agenda which, in turn, causes them to go through essentially the same process.

The knowledge integration subsystem for frames is basically an extension of the *classification* algorithm developed for the NIKL representation language. The NIKL classifier correctly inserts *new* frames into their proper spot in a taxonomy, by finding the most specific set of concepts whose definitions *subsumed* the definition of the new concept. The KREME classifier was designed to additionally allow existing concepts and roles to be modified and and then *reclassified*, so that the effects of redefinitions are automatically propagated throughout the entire frame network. This was accomplished by redesigning the original NIKL classifier to take advantage of the meta-level descriptions of KREME Frames and implementing the new classifier using the dependency directed agenda mechanism of the overall knowledge integrator.

6.1 The Frame Classifier

The remainder of this section will give a brief description of the frame classification part of the knowledge integrator, which is the most completely developed portion of the system. For a formal description of the NIKL classifier algorithm see [14, 15]. For a more complete description of a somewhat simpler classifier for an editing environment, see [1].

The frame classifier works in essentially two stages, starting from a *concept or role definition*, as supplied by the editor or read from a file. The first stage, called *completion*, refers to the basic inheritance mechanism used by KREME Frames to install all inherited features of a concept or role in its internal description. The completion algorithm, when given a set of defined parents and a set of defined features for an object determines the full, logically entailed set of features of that object. The second stage is the actual classification or reclassification of a role or concept. That is, the determination of the complete, most specific set of parents of the object in its respective subsumption hierarchy.

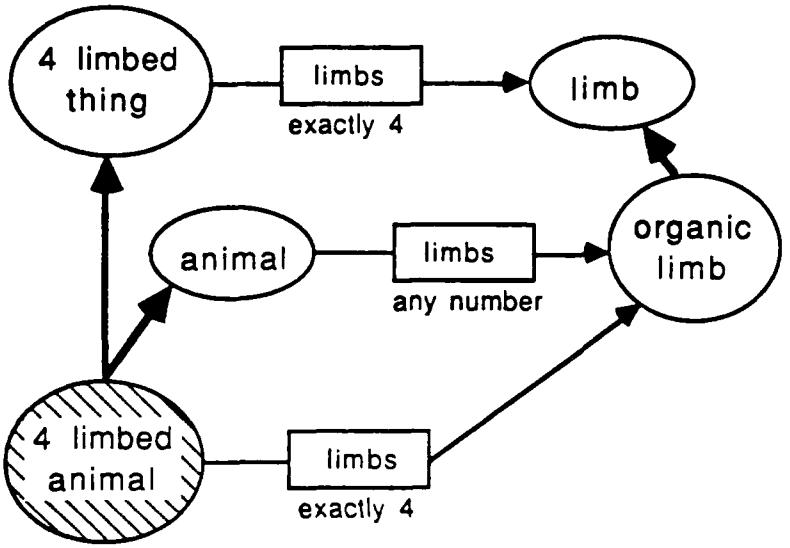
6.1.1 Completion

The completion algorithm is broken up into modular chunks that correspond to the decomposition of the frame language. There is a distinct component that deals with slot inheritance, another component that deals with disjoint class inheritance, a third that deals with slot equivalence inheritance and so on. This organization makes it quite straightforward to extend the language with new features that handle inheritance in different ways.

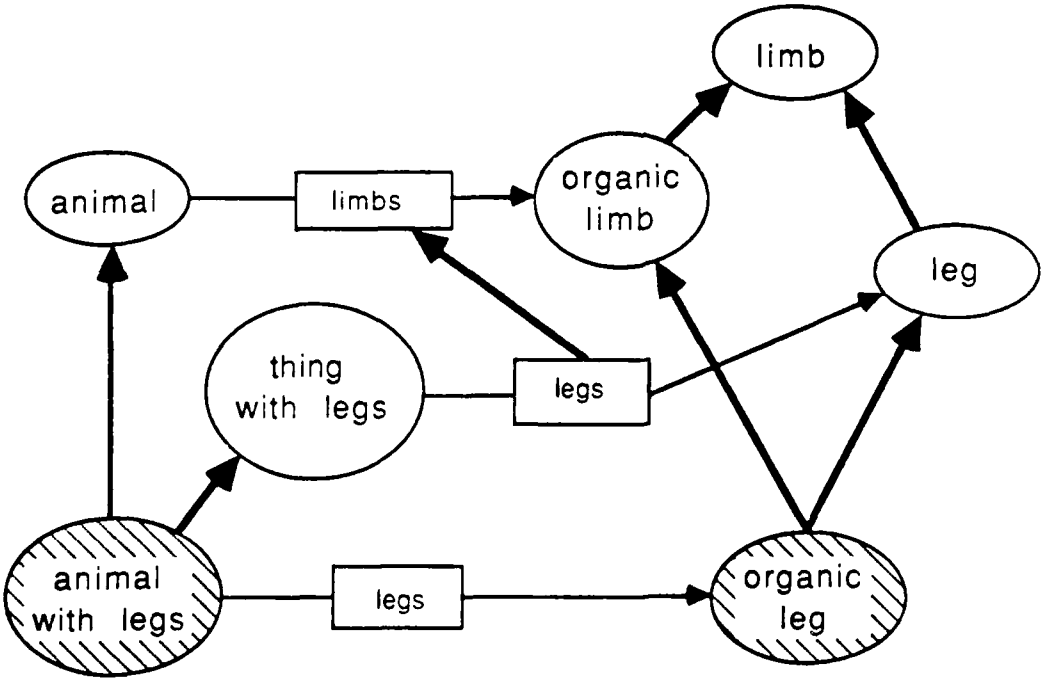
Figure 6-1 shows some of the complexities of slot inheritance. In 6-1A, the most specific *value restriction* for the slot LIMBS at 4-LIMBED-ANIMAL is inherited from one parent (ANIMAL) while the most specific *number restriction*, EXACTLY 4, is inherited from 4-LIMBED-THING. The completion algorithm determines that the restriction for the role LIMBS at the concept 4-LIMBED-ANIMAL must be EXACTLY 4 LIMBS.

Figure 6-1B shows one case for which the effective value restriction must logically be the conjunction of several concepts. Since ANIMAL-WITH-LEGS is both an ANIMAL, and a THING-WITH-LEGS, all of its LIMBS must be both ORGANIC-LIMBs and LEGs. If the concept ORGANIC-LEG, specializing both ORGANIC-LIMB and LEG, exists when ANIMAL-WITH-LEGS is being classified, the integrator will find it and make it the value restriction of the slot LEGS at ANIMAL-WITH-LEGS. If it does not exist, the integrator stops and asks if the user would like to define it (that is, define a concept that is both an ORGANIC-LIMB and a LEG).

Figure 6-1: Two Examples of Slot Completion



Inheriting different number and value restrictions.



Conjoined Value Restrictions.

6.1.2 Classification

The second stage of the frame classification algorithm finds all of the *most specific subsumers* of the concept being defined of redefined. This is the actual *classification* stage, and is essentially a special-purpose tree walking algorithm.

The basic classifier algorithm takes a completed definition (that is, a definition plus all its effective, inherited features) and determines that definition's single appropriate spot in the lattice of previously classified definitions. The result of a classification is a unique set of the most specific objects that subsume the definition and a unique set of the most general objects that are subsumed by the definition. When the classified definition is installed in the lattice all the concepts that subsume its features will be above it in the lattice and all the concepts that are subsumed by its features will be below it.

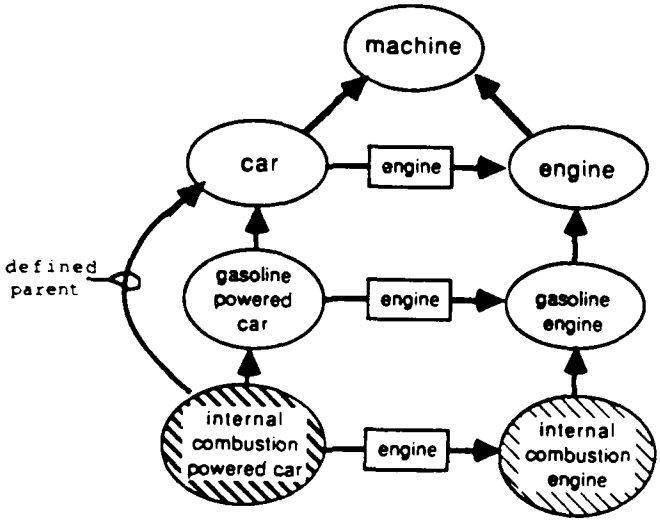
The classifier is built around a modularly constructed subsumption test that compares the completed sets of features of two objects. The object being classified is repeatedly compared to other, potentially related, objects in the lattice to see whether its completed definition subsumes or is subsumed by those other objects. For one definition to subsume the other, its full set of features must be a subset of the features of the other. As with completion, subsumption testing is partitioned by feature type (i.e slot, disjoint-class etc). One object subsumes the other when all of its individual feature-type subsumption checks return EQUIVALENT or SUBSUMES, and there is at least one vote for SUBSUMES. The advantage of this kind of modular organization is extensibility. If a new feature type is added to the language one need only define a subsumption predicate for that feature, and objects having that feature will be appropriately classified.

6.2 An Example of Reclassification

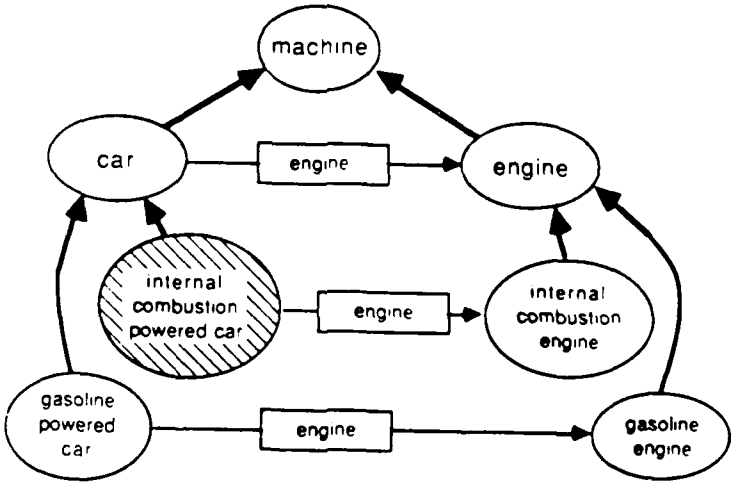
The power of frame reclassification in an editing environment can be illustrated with the following relatively simple example. Suppose a knowledge base developer had defined both GASOLINE-POWERED-CAR and INTERNAL-COMBUSTION-POWERED-CAR as specializations of CAR, but had inadvertently defined INTERNAL-COMBUSTION-ENGINE as a kind of GASOLINE-ENGINE. In this situation, the classifier would deduce that INTERNAL-COMBUSTION-POWERED-CAR must be a specialization of GASOLINE-POWERED-CAR, as shown in figure 6-2A, since the former restricted the role ENGINE to a subclass of the latter's restriction of the same role.

Redefining INTERNAL-COMBUSTION-ENGINE as a kind of ENGINE (rather than a GASOLINE-ENGINE), and then reclassifying, causes all of INTERNAL-COMBUSTION-ENGINE's dependents to also be reclassified,

Figure 6-2: An Example of Reclassification



A. Before Reclassification



B. After Reclassification

including INTERNAL-COMBUSTION-POWERED-CAR. Since GASOLINE-ENGINE no longer subsumes INTERNAL-COMBUSTION-ENGINE, the restrictions for GASOLINE-POWERED-CAR no longer subsume those of INTERNAL-COMBUSTION-POWERED-CAR, and the classifier therefore finds that GASOLINE-POWERED-CAR does not subsume INTERNAL-COMBUSTION-POWERED-CAR. This is shown in figure 6-2B.

The combination of inconsistency detection during the completion phase and the automatic propagation of classification changes that occurs during reclassification makes KREME a powerful and extremely useful tool for knowledge base development and refinement. Since the effects of reclassification are *immediately* made apparent to users via the dynamically updated graph of the subsumption lattice, they sometimes find that the definitions they have provided have some unanticipated logically entailed effects on their taxonomy. Sometimes these effects are surprising, although correct. Other times, they lead to changes and additions which make the knowledge base more complete and correct.

6.3 Using the Knowledge Integrator to Partition and Merge Knowledge Bases

6.3.1 Load/Merge

Perhaps the single most important use for the Knowledge Integrator is to enable orderly merging of independently developed knowledge bases. The process of loading one knowledge base into another is made somewhat involved by the need to merge and/or split and rename concepts that have the same name in both networks.

There are a number of complex cases to deal with. The simplest case occurs when two definitions of the same concept have different but complementary attributes. The KREME merge logic simply forms the union of the attributes of both concepts and edits all pointers to either concept so that they point to the new, enriched concept. (See Figure 6-3.)

A somewhat more complex case occurs when slots shared by both concepts are given different restrictions. (See Figure 6-4.) The system chooses the most specific restriction for the slot.

If concepts with the same name have properties that make it impossible to merge them -- that is, the identical names really stand for *different concepts in the two knowledge bases* (6-5), then the system will inform the user of this fact and ask the user for a **new name** for one concept.

The user has some control over this entire process and can set switches which cause the system to always query when it finds two concepts with the same name, always merge concepts if it can, or never merge concepts, keeping the knowledge bases distinct.

6.4 Saving and Partitioning Knowledge Bases

Any time during the development of a knowledge base, the user can save the entire developing knowledge base to a disk file. This is a useful feature when developing small knowledge bases or working on a piece of a knowledge base that will later be merged into a larger whole.

Another useful facility is KREME's ability to partition a knowledge base along user-designated lines and save the partitions in distinct files. This is accomplished by allowing the user to designate a set of seed concepts. KREME will then create and save a partition of the entire knowledge base, based on the seeds. In an oversimplifed sense, the partition consists of the seeds, all specializations of the seeds, and all the concepts that the seeds either directly or indirectly depend on. This facility can be used to break up a single knowledge base into several overlapping subcomponents.

6.5 Using Merge and Partition to Build Larger Knowledge Bases

Taken together, the merge and partition facilities suggest an approach that we think will prove to be an extremely powerful paradigm in the building of very large, very complex knowledge bases. When a knowledge base grows to a size at which it becomes difficult to deal with in its entirety, the partition/save facility can be used to divide it into several overlapping logical subcomponents, each of which is a full scale, consistent knowledge base in its own right.

These multiple, smaller knowledge bases can be worked on independently of each other with full confidence that the loader/merger can put the independently built subcomponents together in an orderly, consistent fashion.

In Figure 6-5, there are two networks. The "ball" in Network 1 stands for a concept that is a kind of round object. In Network 2, the name "ball" stands for a kind of formal dance. These are differenct concepts with unmergeable properties. In both networks, *Event* and *Object* would be defined to be disjoint. In this case, the Merger would ask the user for a new name for one of the concepts and would keep them distinct.

Figure 6-3: Example One : Merging with Nonoverlapping Attributes

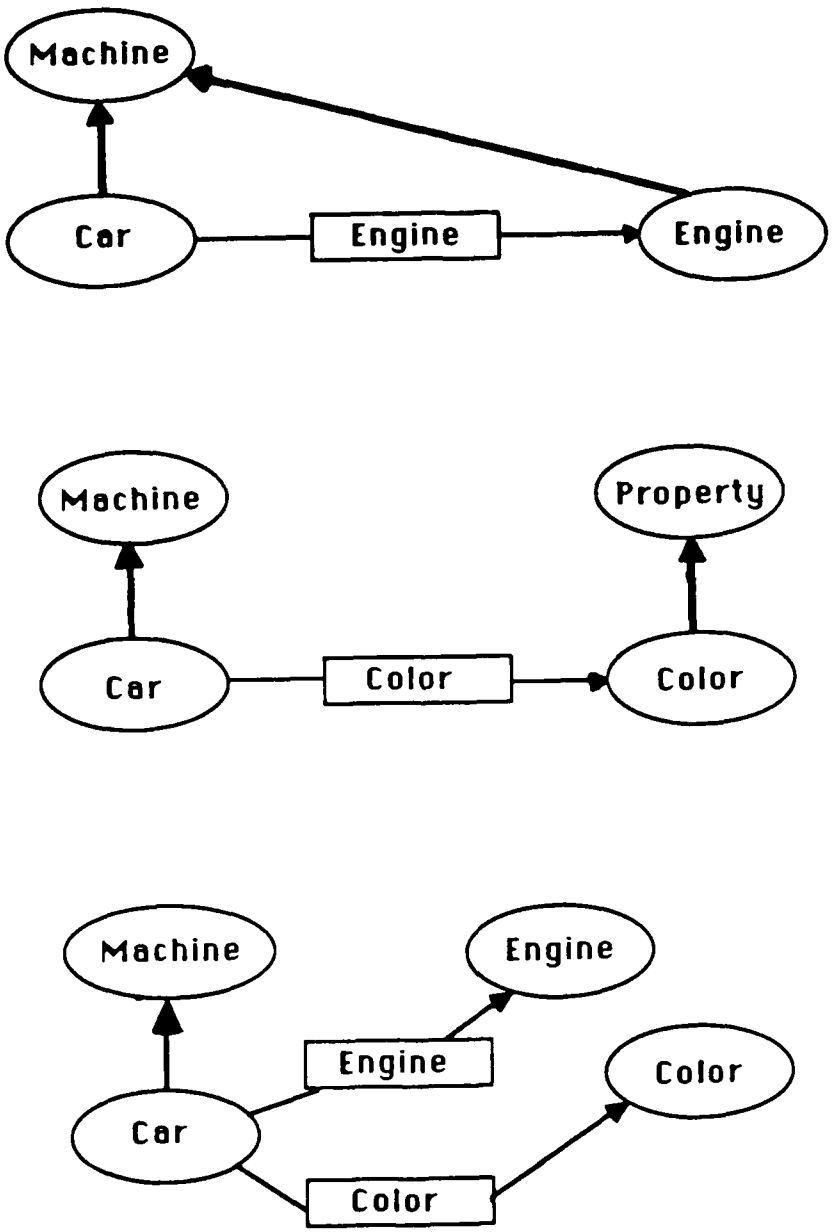


Figure 6-4: Example Two: Overlapping but Compatible Properties

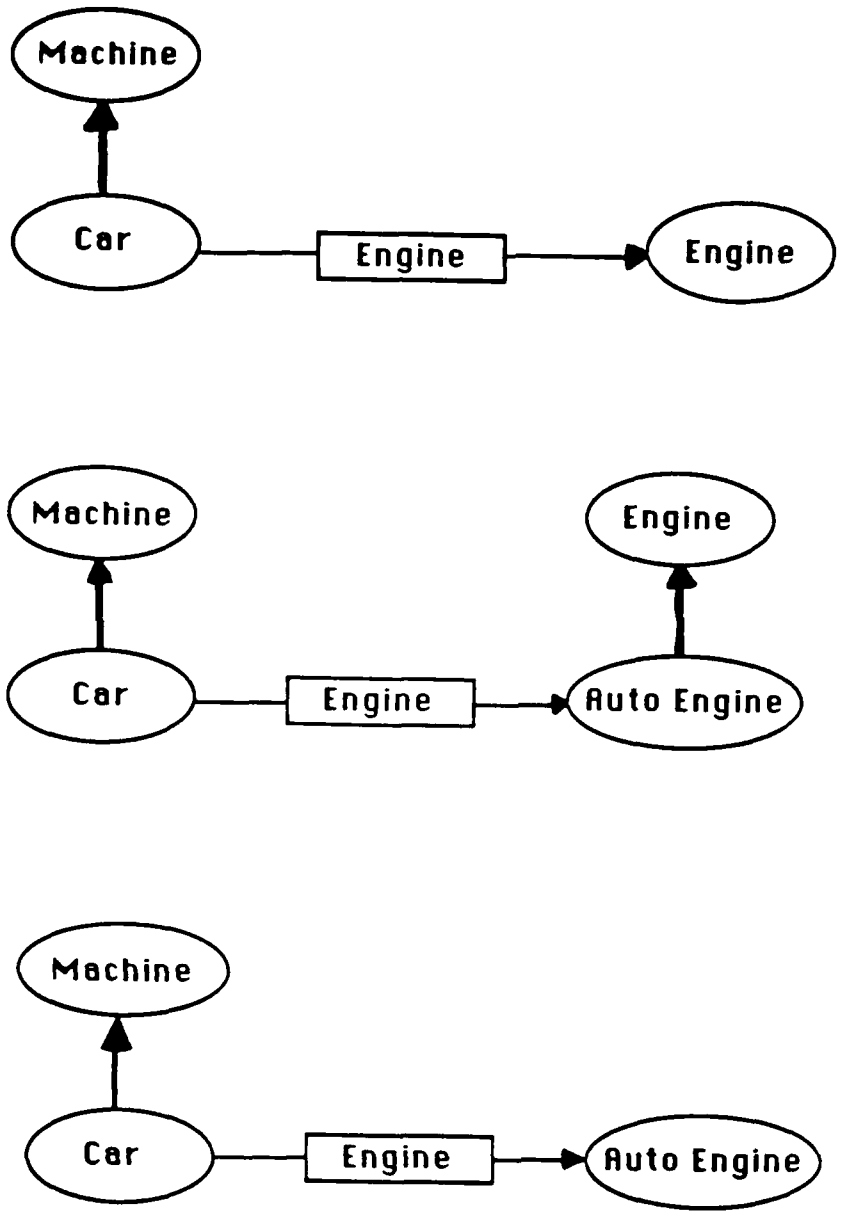
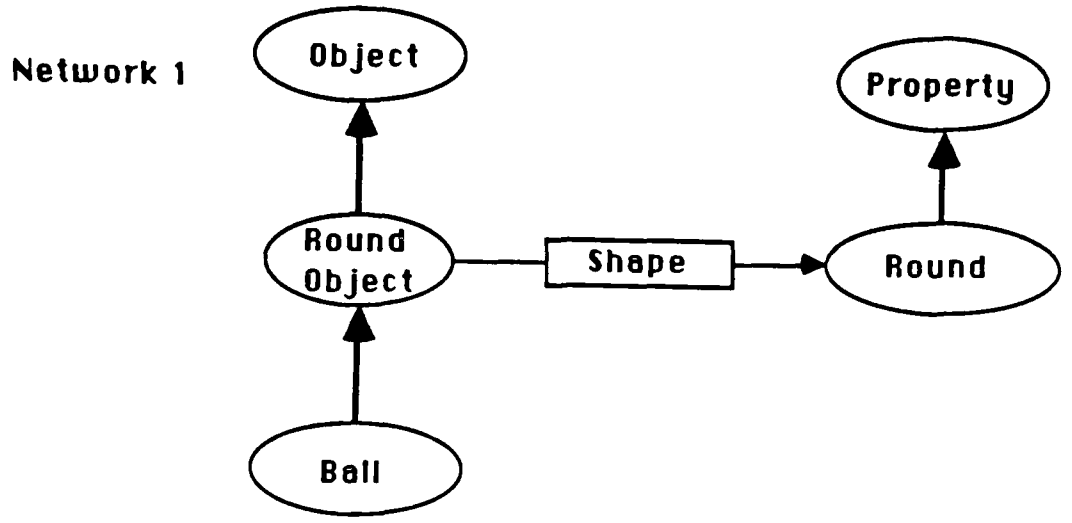
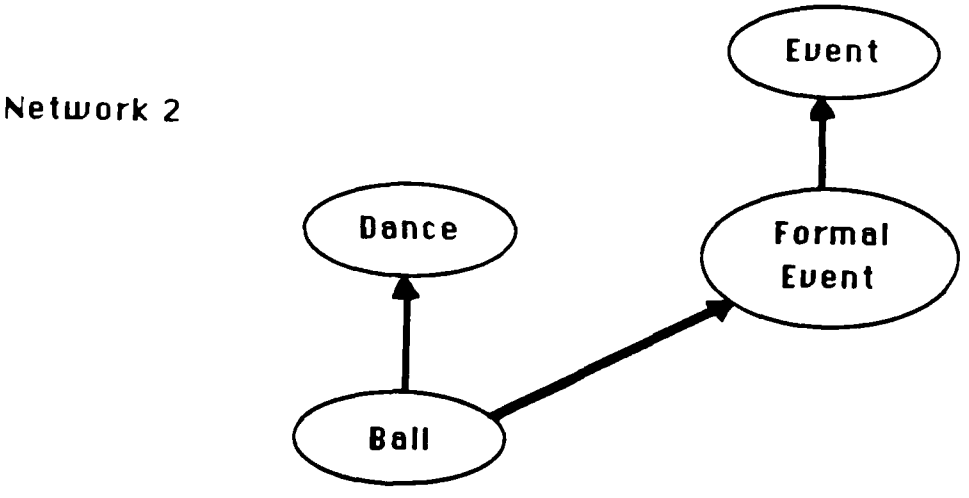


Figure 6-5: Example Three: Unmergeable Concepts





7. Editing Behavioral Knowledge

KREME embodies a set of mechanisms for representing and editing behavioral knowledge. One mechanism involves associating behaviors with frames. Since frames can also be associated with *flavors*, behaviors have been implemented so that they can be compiled into flavor methods.

A click of a mouse button and the *tabular features window* in the *main concept view* is turned into the toplevel behavior editor. All behaviors currently defined for the concept are shown. Each has a name and a type. There are three types of behaviors currently allowed: Rules, Procedures, and Methods. Existing behaviors can be edited or new ones defined. A modified form of the Symbolics *flavor examiner* can be accessed to show various useful information about method combination and derivation.

Methods are simply flavor methods. Editing a method throws up a text editor window which can be interacted with in normal editing style or in structure editing style. Editing or inputting a new rule packet accesses the Rule Editor. Editing or inputting a new procedure accesses the Procedure Editor.

7.1 Editing Rules

The rule language used by KREME is a language called FLEX [16], based in large part on the LOOPS rule language. FLEX allows rules to be defined in *rule packets*, which organize sets of rules that are meant to be run together. In the KREME environment, rule packets can be attached to concepts, just as if they were functional methods. In addition, they may be inherited by more specialized concepts. FLEX incorporates a mechanism for dealing with uncertainty, based on EMYCIN [19]. The FLEX runtime environment also provides an elementary history and tracing mechanism, and an explanation system that produces pseudo-English explanations from rule traces. For efficiency, FLEX also provides a means for rule packets to be compiled as LISP code, and run without the rule interpreter present.

The KREME rule editor is built on top of the KREME structure editor. One defines and edits rules by specifying and filling out portions of rule *templates*. The user refines these templates either by using the mouse to copy parts of existing rules or by pointing at slots to be filled and typing in the desired values. Once a rule-set has been developed, the rule editor provides commands to run packets and debug them. It can also generate traces or rule histories paraphrased in pseudo-English. Mechanisms are also provided for deleting and reordering rules, and loading and saving them from files. The rule editor is shown in figure 7-1.

The rule editor is also tied to the KREME's knowledge integration subsystem. At present, all references to slots of frames made in rules are checked for validity by the knowledge integrator. If invalid, the user is alerted and may switch, if necessary, to editing the associated frame. If the problem was simply that he/she named a non-existent slot, a valid one may be selected from a menu. In the near future, the knowledge integrator will also check such cross-references in the opposite direction, as when a slot referred to by some rules is deleted or changed in the frame editor.

KREME at present edits rules in the FLEX [16] rule language. In FLEX, rules come in *rule packets*, and the KREME Rule Editor edits an entire packet at one time. Rule packets provide a way to organize rules.

The forward chaining rule packets come in four varieties, indicating the type of control mechanism used for rule firing.

- **do-1-rule-packets** execute the first rule whose test succeeds.
- **do-all-rule-packets** execute all rules whose tests succeed.
- **while-1-rule-packets** repeatedly test all rules, firing one, until no tests succeed.
- **while-all-rule-packets** repeatedly fire all rules whose tests succeed, until none succeed.

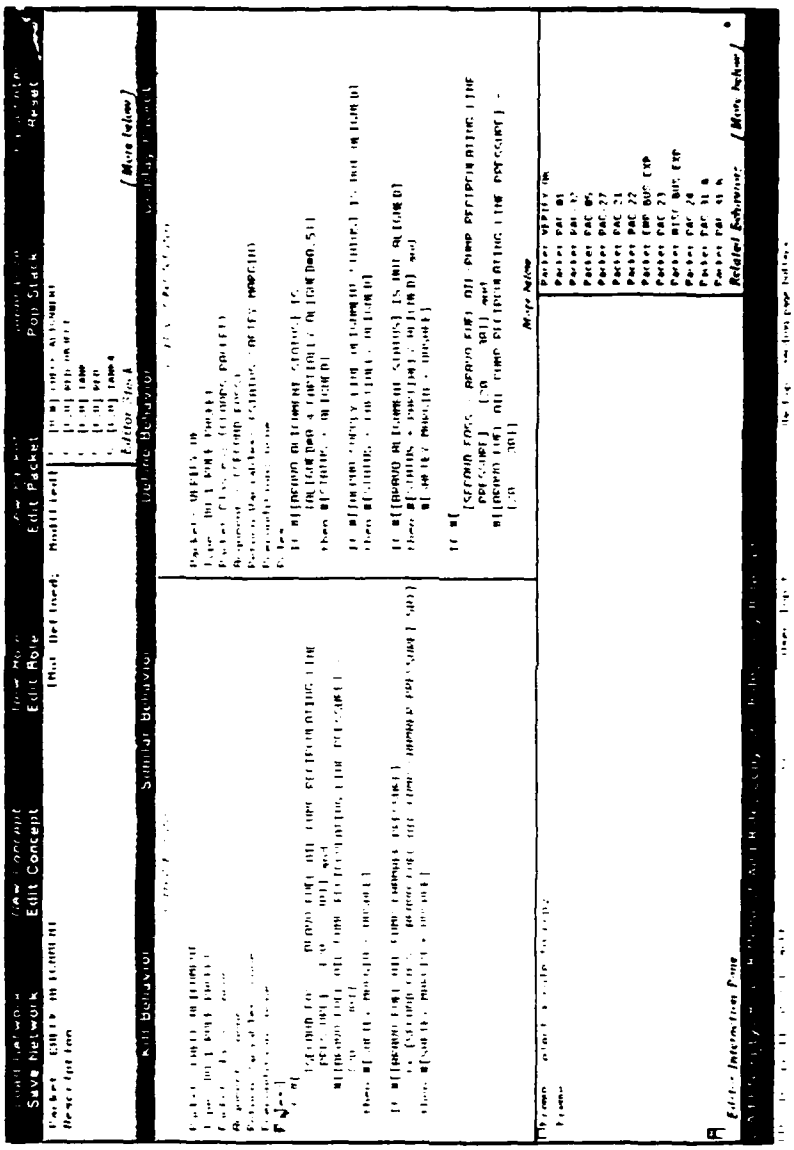
Rule packets are connected to KREME frame systems or other data contexts by specifying an *access environment*. An access environment is an object that receives messages dealing with the accessing of values for references in the rules. It handles all messages to get or set the values of variables and their confidences.

7.2 The KREME Rule Editor

Rules are defined and edited by specifying and filling out portions of rule *templates*. To refine these templates either use the mouse to copy parts of existing rules or point at slots to be filled and type in the desired values.

There are also commands to run packets and debug them and to generate traces or rule histories paraphrased in pseudo-English, and delete rules and reorder rules, and load and save rules from files.

Figure 7-1: The KREME Rule Editor



branching or iteration; the mechanisms for procedural abstraction, specialization, and path or reference reformulation that formed the heart of the language seemed to form the kernel of an extremely useful representational facility.

The KREME representation language family includes a descendant of the STEAMER procedure language, built using KREME's library of knowledge representation primitives. Each KREME procedure has a *name*, a *description*, an *action* that the procedure is meant to accomplish, a list of *steps*, and a list of *ordering constraints* that determine the partial ordering of the steps. *Steps* have an *action* and an *object* which names the conceptual class of things that step acts upon. Procedures are attached to specific frames and can be "compiled" into flavor methods.

Each step in a procedure may either be a primitive action or another procedure. If the object of a step defines a procedure for the action of that step then this procedure is said to be a sub-procedure of the enclosing procedure. For example, the ALIGN procedure attached to the concept SUCTION-LINE could have a step ALIGN <PUMP>. If the concept CENTRIFUGAL-PUMP, which is the object of this step for SUCTION-LINEs, defined a procedure for the action ALIGN, then the step ALIGN <PUMP> could be expanded into the steps of the procedure for aligning a centrifugal pump.

7.4.1 Procedural Abstraction and Structure Mapping

For knowledge acquisition purposes, it would be very useful if procedures were represented in an abstraction hierarchy like that for frames. In a strong sense, it seems difficult to define exactly what it means for one abstract procedure to subsume another. However, from an acquisition standpoint, much power can be gained by allowing abstract procedures to form templates upon which more specific procedures can be built, and eventually providing tools for automatic plan refinement like those found in NOAH [13]. For example, if you have some idea about how to grow plants in general, and you want to grow tomatoes, you will use your knowledge about growing plants in general as a starting point for learning about growing tomatoes. The final procedure for growing tomatoes will include some (presumably more detailed) versions of steps in the more general procedure, and may also include steps that are analogous to those used in growing other plants for which more detailed knowledge exists.[5]

The KREME Procedures editor has a mechanism for building templates of new procedures out of more abstract procedures. When a new procedure is being defined at a concept, the procedural abstraction function determines whether any of that concept's parents have a procedure for accomplishing the same action. If so, an initial procedure template is built by combining the steps and constraints of all the inherited, more abstract procedures. The paths

[5]For a detailed discussion of related issues see Carbonell [4] on derivational analogical planning

(objects) of the steps are adjusted, using the concept's slot equivalences, to use "local" slot names, as much as possible. As yet this facility does not have the ability to do detailed reasoning with constraints on steps, as NOAH does. We expect to greatly expand this capability during Phase Two of the project.

8. Knowledge Extension

One task faced by knowledge engineers is getting experts to express *generalizations* about their domains of expertise. While much of the detailed information about particular problems can be accessed and represented by looking at specific examples and problems, the expert's abstract classification of problem types and the abstract features he uses to recognize those problem types are less directly available. Experienced knowledge engineers are often able to discover and define useful generalizations which experts perceive as relevant to their own reasoning processes. The experts may then suggest improvements, related generalizations, or more abstract generalizations.

Our initial experiment in knowledge-base extension in Phase 1 has been the development of a *frame generalization* algorithm. Our current generalizer finds potentially useful generalizations by searching for sets of concept features that are shared by several unrelated concepts.

When the generalizer finds a set of at least k features shared by at least m concepts, where k and m are user-settable parameters, the system forms the most specific concept definition that would enclose all of the features but would still be more general than any concept in the set. Since our simple algorithm has no other external notion of "interestingness" it simply displays this potential new concept definition to the user. For example, given three concepts that are all ANIMALs and independently define the slot WINGS, the generalizer would suggest forming a specialization of ANIMAL with the slot WINGS, that these concepts would all specialize. If the user wanted to introduce this concept, he would respond by naming the new generalization (e.g., FLYING-ANIMAL), which would then be classified and integrated with the network. The features that are enclosed by this new, more general concept, are automatically removed from each of the more specific concepts being generalized.

7.3 The Rule Editor View

Many of the windows in the **Rule Editor View** should be familiar by now. The complete list is as follows.

1. **Global Command Window** displays global commands that can be selected by the user. In this example, the user has used the mouse to select **Edit Packet**. The user's selection is highlighted.
2. **State Window** displays the name of the packet, the network it is associated with, and other useful information.
3. **Editor Stack Window** displays the names of the items recently edited and some information on their current state. Items in the editor stack window can be selected for editing with the mouse.
4. **Behavior Command Window** is a menu of commands that apply to Rules and Rule Packets. (*Behavior* is another term for rule packets, or functional methods on instances of concepts.)
5. **Current Edit Item Window** displays the item that has been selected for editing.
6. **Dispay Related Items Window** allows the user to view other rule packets and scroll through them. Rules and parts of rules can be copied from the Scroll Window into the Current Edit Item Window.
7. **Editor Interaction Window** displays screen prompts and user input. The user's edits are made in this window and then displayed in the Current Edit Item Window.
8. **Related Behaviors Window** displays an index of other rule packets that are related to the one currently being edited. With the mouse, the user can rapidly scroll through this index and select a related rule packet for viewing or editing.

To get into the Rule editor use the **New Packet** or **Edit Packet** command in the **global command window**.

Thereafter, the structure editor can be used in much the same way the Macro Structure Editor is used to edit concepts. The **Rule Structure Command Menu** contains the commands:

- **Define Behavior** is similar to **Classify Concept**. It makes the definition of the packet permanent, and allows it to be run or attached to a concept.
- **Similar Behavior** - Creates a packet with the same rules, etc. but gives it a new name, and presents it to be edited to make it different.
- **Kill Behavior** - Kills the definition of this packet.
- **Display Packet** - Displays the packet in the **Display of Related Items Window**.

When a whole rule packet is outlined, the user can choose to **Edit Packet** (L:), or (R:) choose from a menu of **Edit Packet**, **Edit Basis** or **Display Lisp Form**.

Other editing commands are found on the keywords and component pieces of packets and rules. For instance, clicking left on **Rule:** places a new (empty) rule in the packet, which can then be filled out by clicking on **IF** to add a new condition (conditions are treated as part of a conjunction) or **THEN** to add a new action. Clicking right gives a menu of **Add (Empty** Rule), **Copy One Rule** from somewhere else into this packet, and **Copy Rule Set** which copies all of the rules from another packet.

Clicking over **Type:** gives the user a choice of the standard types of rule packets, described above.

Packet Classes: allows the user to specify a flavor to be mixed into the packet. **Arguments:** and **Return Variables:** each allow the user to add a new one (L:) or choose from a menu of **Add One**, **Add Several**, **Edit** and **Replace**.

When a whole rule is outlined, clicking left will be replace the rule with another rule that the user points at. Clicking right gives a menu of **Replace Rule**, **Edit Attributes** and **Delete Rule**.

Whenever expressions appear (after the word **Precondition:**, or as parts of conditions or actions), the user may **Replace** the expression (L:), or choosing from a menu (R:) of

- **Replace** the expression with another one.
- **Edit** the expression as text.
- **Delete** the expression.
- **Add Before** another expression (copied from somewhere by pointing).
- **Add After** another expression.
- **Exchange** two expressions positions.
- **Parenthesize** a set of expressions together.
- **Deparenthesize** an expression into piecies.
- **Evaluate** the expression in the current context.

7.4 Procedures in the KREME Environment

An obvious weakness of many knowledge representation languages is their inability to handle declaratively expressed knowledge about procedures as partially ordered sequences of actions, particularly if that knowledge is represented at multiple levels of abstraction. Although a number of systems have been developed that do various forms of planning, [5, 12, 13, 18], most have not encoded their plans in an entirely declarative or inspectable fashion. Certainly the current generation of expert system tools does not provide mechanisms geared to the description of this kind of knowledge. Although it is clear that much of an expert's knowledge about a domain is about procedures and their application, little work has been done on devising ways to capture that information directly.

The STEAMER project [21] began to address the issue of declarative representations for procedures in the course of developing a mechanism to teach valid steam plant operating procedures. The representation system developed for this task had to be directly accessible to the students who were the system's users, and it had to serve as a source of explanations when errors were made. STEAMER was able to describe these procedures, decompose them, show how they were related to similar procedures and, in general, deal with them at the "knowledge level" [10] rather than as pieces of programs or rule sets. Although the syntax of the language was quite primitive, with no provisions for

9. Conclusion

The goal of the BBN Laboratories Knowledge Acquisition Project is to build a versatile experimental computer environment for developing and maintaining large knowledge bases. We are pursuing this goal along two complementary paths. First, we have constructed a flexible, extensible, Knowledge Representation, Editing and Modeling Environment in which different kinds of representations (initially frames, rules, and procedures) can be used. We are now using this environment to investigate acquisition strategies for a variety of types and combinations of knowledge representations. In building and equipping this "sandbox", we have been adapting and experimenting with techniques which we think will make editing, browsing, and consistency checking for each style of representation easier and more efficient, so that knowledge engineers and subject matter experts can work together to build with significantly larger and more detailed knowledge bases than are presently practical.

The second aspect of our research plan is the development of more automatic tools for knowledge base reformulation and extension. An important part of this endeavor is the discovery, categorization and use of explicit knowledge about knowledge representations: methods for viewing different knowledge representations, techniques for describing knowledge base transformations and extrapolations, techniques for finding and suggesting useful generalizations in developing knowledge bases, semi-automatic procedures of eliciting knowledge from experts, and extensions of consistency checking techniques to provide a mechanism for generating candidate expansions of a knowledge base.

We are attempting to provide a laboratory for experimenting with new representation techniques and new tools for developing knowledge bases. If we are successful, many of the techniques developed in our laboratory will be adopted by the comprehensive knowledge acquisition and knowledge representation systems required to support the development and maintenance of future AI systems.

10. Appendix A: Test Plan

10.1 Loading KREME from Cassette Tape

Each site can test KREME by loading KREME from tape according to the directions in this Appendix and then editing the sample networks provided on the tape. Once KREME has been loaded, the document *KREME: A User's Introduction* BBN Report No. 6508 provides instructions on how to edit and create knowledge bases using KREME.

KREME requires a Symbolics machine with Genera-7.0 already installed and with at least 18000 blocks free in its FEP. If your machine has no tape drive, you will have to read the tape on another machine that does have one and then transmit the bands to your machine. (See section 10.2.) We will use the terms **destination machine** and **tape drive machine** to refer to these two machines. Note that you must have at least 18000 blocks free on the destination machine's FEP as well as having at least 18000 blocks free on the FEP of the machine with the tape drive.

10.1.1 Loading the FEP Files

There are four FEP Files on the tape. Your machine may already have **inc-7-0G1-from-Genera-7-0.load**. If so, do not create a FEP file for that file and do not load it from the tape.

Log in to the machine and create three (or four) FEP files in the following way:

```
Create FEP File inc-7-0G1-from-Genera-7-0.load  1290
Create FEP File inc-BBN-from-inc-genera-7-0G1.load  5600
Create FEP File Kreme-from-Boot7.load  9580
Create FEP File Kreme.boot  1
```

Log out and halt the machine.

Put the FEP Files tape in the tape drive.

Type the following to the FEP:

```
scan V127-disk
```

(This teaches the FEP about disk restore.) Then type

```
disk restore
```

The machine will then ask if you've done **Set Disk Type**. Answer **Y**[1] The machine then asks if you want to restore the FEP files on the tape. In each case answer **Y** and press carriage return. (If you already have the first band on your system, answer **N** for that band.) In each case, the system will then prompt

[1] If the disk is new and has not been initialized, see your local system wizard.

```
file to restore?
```

Accept the default file name by pressing carriage return.

The machine displays numbers as it reads from the tape. The machine then asks about the other files in turn. Each time, answer Y to restore the file and then press carriage return to accept the default file name.

10.1.2 Editing the FEP Files

Now you must edit the file Kreme.boot to set the CHAOS address correctly. To do this, boot the machine (using a boot file other than Kreme.boot) and edit Kreme.boot. Change the line containing the CHAOS address to set it to the address of the destination machine. You can get the correct CHAOS address for the destination machine from the system manager or by looking at the address in another .boot file on the destination machine.

You must also edit the Load Microcode line in Kreme.boot so that it contains the number of the microcode version on the host. To determine that number, ask the system manager or look at a .boot file that boots a 7.0 world.

Now log out and halt the machine.

10.1.3 Booting KREME

Type the following to the FEP:

```
Boot Kreme.boot
```

Because the band is being booted at a site other than the site at which it was built, the machine will ask you if the site is still BBN. Answer NO and the machine will name itself **DIS-LOCAL-HOST**.

If the machine has identity problems (It thinks it is still at BBN), the simplest way to deal with them is to unplug the ethernet before booting Kreme.boot. See your local system wizard if you want a more elegant solution.

Once the boot is complete, you'll have a KREME window with the

```
KREME:
```

prompt. Now get to a Lisp Listener via

```
<select>L
```

Then log in with the command

```
(si:login-to-sys-host)
```

Logging in in this way avoids interacting with the BBN system accounting software. Then load the carry-tape with the command

```
(tape:carry-load)
```

The carry-tape contains two sample KREME networks, mech-net.lisp and org-net.lisp. You will have to choose a place on your machine to store these files.

You are now ready to use KREME with the help of *KREME: A User's Introduction*. Try loading a sample network from one of the files you read off the carry-tape.

10.2 For Machines with No Tape Drive

First, load the FEP Files from the tape onto the tape drive machine by following the instructions in section 10.1.1. Then boot that machine, using a boot file other than Kreme.boot. Then transmit the FEP Files to the destination machine by typing the following to a Lisp Listener: (Answer Y when the system asks if you really want to.)

```
(si:transmit-band "fep0:>inc-7-0G1-from-genera-7-0.load"
                                        destination-machine)

(si:transmit-band "fep0:>inc-bbn-from-inc-genera-7-0.load"
                                        destination-machine)

(si:transmit-band "fep0:>Kreme-from-boot7.load"
                                        destination-machine)

Copy File fep0:>Kreme.boot  destination-machine|fep0:>Kreme.boot
```

You are now finished using the machine with the tape drive. You may delete the KREME files on that machine before going to the destination host.

Now continue with the instructions in section 10.1.2.

Bibliography

1. Balzac, Stephen R. A System for the Interactive Classification of Knowledge. M.S. Thesis, M.I.T. Dept of E.E. and C.S., 1986.

2. Bobrow, D., Winograd, T. and KRL Research Group. Experience with KRL-0: One cycle of a knowledge representation language. Proceedings of the Fifth International Joint Conference on Artificial Intelligence. Cambridge, MA., August, 1977.

3. Brachman, R.J., Fikes, R.E., and Levesque, H.J. "Krypton: A Functional Approach to Knowledge Representation". *IEEE Computer, Special Issue on Knowledge Representation* (October 1983).

4. Carbonell, Jaime G. Derivational Analogy: A theory of reconstructive problem solving and expertise acquisition. In *Machine Learning: Volume II*, Michalski, R. S., Carbonell, J. G. and Mitchell, T. M., Ed., Morgan Kaufmann Publishers, Inc., Los Altos, CA. 1986, pp. 371-392.

5. Ernst, G.W. and Newell, A.. *GPS: A Case Study in Generality and Problem Solving*. Academic Press. New York, 1969.

6. IntelliCorp. *KEE Software Development System*. IntelliCorp, 1984.

7. Keene, Sonya E. and Moon, David. Flavors: Object-oriented Programming on Symbolics Computers. Symbolics, Inc.

8. Carnegie Group, Inc.. *KnowledgeCraft*. Carnegie Group, Inc., 1985.

9. Moser, Margaret. An Overview of NIKL. Section of BBN Report No. 5421, Bolt Beranek and Newman Inc., 1983.

10. Newell, A. "The knowledge level". *AI Magazine* 2, 2 (1981), 1-20.

11. Roberts, B. and Goldstein, I. P. The FRL Manual. A.I. Lab. Memo 409, M.I.T., 1977.

12. Sacerdoti, E. E. "Planning in a Hierarchy of Abstraction Spaces". *Artificial Intelligence* 5, 2 (1974), 115-135.

13. Sacerdoti, Earl D. A structure for plans and behavior. 109, SRI Artificial Intelligence Center, 1975.

14. Schmolze, J. and Israel, D. KL-ONE: Semantics and Classification. In *Research in Knowlege Representation for Natural Language Understanding, Annual Report, 1 September 1982 to 31 August 1983*, BBN Report No. 5421, 1983.

15. Schmolze, J.G., Lipkis, T.A. Classification in the KL-ONE Knowledge Representation System. Proc. 8th IJCAI, 1983.

16. Shapiro, Richard. FLEX: A Tool for Rule-based Programming. 5643, BBN Labs., 1984.

17. Sidner, C.L.; Bates, M.; Bobrow, R.J.; Brachman, R.J.; Cohen, P.R.; Israel, D.J.; Webber, B.L.; and Woods, W.A. Research in Knowledge Representation for Natural Language Understanding: Annual Report. BBN Report No. 4785, Bolt Beranek and Newman Inc., November, 1981.

18. Stefik, Mark. "Planning with Constraints: MOLGEN". *Artificial Intelligence 16*, 2 (1981), 111-169.

19. van Melle, W. A domain independent production-rule system for consultation programs. Proceedings of IJCAI-6, August, 1979, pp. 923-925.

20. Vilain, Marc. The Restricted Language Architecture of a Hybrid Representation System. Proceedings, IJCAI-85, International Joint Conferences on Artificial Intelligence, Inc., August, 1985, pp. 547-551.

21. Williams, M., Hollan, J., and Stevens, A. "An Overview of STEAMER: An Advanced Computer-Assisted Instruction System for Propulsion Engineering". *Behavior Research Methods and Instrumentation 14* (1981), 85-90.

END

DATED

FILM

9-88

DTIC